Süsteemprogrammeerimine keeles C

Kuues loeng

milles räägime mis on static ja mis on extern, teeme ühe listinäite, räägime union andmetüübist, objektide mälusse asetamise nõuetest, vaatame bittoperaatoreid ning räägime rekursioonist

Eelmises osas(Kahemõõtmelised massiivid)

Kahemõõtmelise massivi saame ühemõõtmelisest ümberadresseerimise teel: elem[i][j] = elem[i*k + j], kus k on j pikkus.

Dünaamilise kahemõõtmelise massiivi jaoks hõivame kaks pointerit nii, et esimene viitab teise massiivi k-kordsetele aadressidele.

Eelmises osas (Pointerite teisendamine)

Pointerite teisendamiseks kasutatakse void tüüpi pointerit, millesse ja millest on võimalik tüüpe alati kadudeta teisendada.

```
void *someptr;
```

Eelmises osas (andmevahetus binaarkujul)

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

Pointer andmetele
Andmetüübi suurus
Andmeühikute arv
Kasutatav stream

Eelmises osas (ajutised failid)

- `FILE* tmpfile();`
- Tagastab pointeri *stream*-ile, mis on avatud nii lugemiseks kui kirjutamiseks
- Ajutine fail kustutatakse pärast selle sulgemist fclose() funktsiooniga või kuni programmist väljumiseni
- Faili asupaik sõltub süsteemi ja standardteegi valikutest.

Eelmises osas (failipointer)

int fflush(FILE *fp); kirjutab failipuhvri tühjaks

int fseek(FILE* fp, long offset, int place);
SEEK_SET, SEEK_CUR, SEEK_END
long ftell(FILE *fp);
void rewind(FILE* fp);

int fgetpos(FILE * fp, fpos_t* pos);
int fsetpos(FILE * fp, const fpos_t* pos);

Tänases loengus

- Listi näide
- Union andmetüüp
- *Alignment*
- Bittoperaatorid
- Rekursioonist

Stringide lugemine (gets, fgets)

- char *gets(char *s);
  - gets loeb standardsisendist stringi.
  - reavahetus asendub \0 märgiga
  - **ära kunagi kasuta!** sa ei saa anda ette puhvri suurust
- char *fgets(char *s, int size, FILE *fp);
  - fgets loeb viidatud puhvrisse stringi failist kuni reavahetuse või EOF-ni. Maksimaalselt *size* tähemärki.
  - reavahetus ei asendu, \0 lisatakse
- EOF puhul tagastub NULL

Stringide kirjutamine (puts, fputs)

- int puts(const char *s);
  - kirjutab stringi standardväljundisse
  - kirjutab \0 asemel reavahetuse \n
  - **ära kunagi valjusti ütle!**
- int fputs(const char *s, FILE *fp);
  - kirjutab stringi viidatud *stream*'i
  - reavahetust \n ega \0 märki ei lisata
- Edu korral tagastub mittenegatiivne arv, vea korral EOF

sprintf(), sscanf()

- Stringide vormindatud lugemine ja kirjutamine

```
int sprintf(char *s, const char *format, ...)
int sscanf(const char *s, const char *format, ...)
```

- fprintf ja fscanf stringiversioonid

```
char str1[]="1 2 3 go", str2[100], tmp[100];
int a,b,c;
sscanf(str1, "%d%d%d%s", &a, &b, &c, tmp); // reads
from its first arg
sprintf(str2, "%s %s %d %d %d\n", tmp, tmp, a, b, c);
printf("%s", str2);
```

```
go go 1 2 3
```

string.h

- Üldinfo
  - stringilibra funktsioonid arvestavad, et \0 lõpetab stringi
  - kui \0 puudub, võib väljund olla "natuke" vale

```
char dst[100], s[]="abc";
s[3] = 'd'; /* '\0' kirjutatakse üle */
strcpy(dst, s); /* oih! */
```

Stringi pikkus

- size_t strlen(const char *s);
  - tagastab stringi pikkuse
  - const sellises deklaratsioonis sümboliseerib, et s ei ole funktsioonisiseselt muudetav
  - size_t on enamasti kas unsigned int või unsigned long

```
size_t strlen(const char *s) {
 size_t n;
 for (n = 0; *s != '\0'; s++)
    n++;
 return n;
}
```

```
size_t strlen(const char *s) {
    const char *t = s;
    while (*t)
       t++;
    return t - s;
}
```

Stringi kopeerimine

- char *strcpy(char *dest, const char *src);
  - kopeerib stringi *src* stringiks *dest*
  - jälgida, et *dest* oleks *src* mahutamiseks piisavalt suur
- char *strncpy(char *dest, const char *src, size_t n);
  - kopeeritakse maksimaalselt n baiti
  - kui on täpselt n, siis \0 jääb lõpust ära
  - kui on vähem kui n, kirjutatakse ülejäänutesse \0

Stringide võrdlemine

- int strcmp(const char *s1, const char *s2);
  - võrdleb stringe omavahel
  - tagastab 0, kui stringid on võrdsed
  - kui s1 > s2, siis tagastab arvu , mis on suurem kui 0
  - ja vastupidi

```
"str1"   >  "str0"
"str1"   == "str1"
"str1"   <  "str2"
```

  - levinud viga:

```
if ( strcmp(a, b) ) ... // VIGA!
```

- int strncmp(const char *s1, const char *s2, size_t n)

Stringide jupitamine

- char *strtok(char *str, const char *delim);
  - tagastab stringi *str* järgmise jupi või NULL, kui neid enam pole
  - argumendina esimeses väljakutses jagatav string, edaspidi NULL
  - delim on string juppe eraldavatest charidest
  - puudusteks on see, et me ei saa teada, mis see eraldusmärk oli ja see, et esimest argumenti muudetakse

strtok() näide

```
void print_tokens(char *line)
{
  char whitespace[]=" \t\f\r\v\n";
  char *token;
  for (token = strtok(line, whitespace); token != NULL;
          token=strtok(NULL, whitespace)) // NULL!
   printf("Järgmine on %s\n", token);
}
```

Ülevaade stringifunktsioonidest (1)

- **char *strcat(s, cs)** Liidab stringi s otsa stringi cs ja tagastab s-i
- **char *strncat(s,cs, n)** Lisab stringi s otsa maksimaalselt n stringi cs märki
- **char *strcpy(s, cs)** Kopeerib cs-i s-iks, ka \0
- **char *strncpy(s,cs)** Kopeerib cs-i maksimaalselt n tähemärki s-iks, ülejäänud määrab \0-deks
- **char *strtok(s,cs)** Leiab stringi s seest cs-i märkidega eraldatud osad
- **size_t strlen(cs)** Tagastab cs-i pikkuse (v.a. \0)
- **int strcmp(cs1,cs2)** Võrdleb cs1 ja cs2, tagastab nullist suurema, võrdse või väiksema arvu vastavalt sellele, kas cs1 on cs2-st vastavalt suurem, võrde või väiksem
- **int strncmp(cs1, cs2, n)** Võrdleb cs1 ja cs2 esimesi n-i tähte omavahel

Ülevaade stringifunktsioonidest (2)

- **char *strchr(cs,c)** Annab esimese c esinemise peale stringis s pointeri
- **char strrchr(cs,c)** Annab viimase c esinemise peale stringis s pointeri
- **char *strpbrk(cs1, cs2)** Otsib stringist cs1 esimese cs2 tähe esinemise ja annab sellele pointeri
  - nt: strpbrk(“Selle lause lõpp on kirjas maja peal aadressil”, “cba”) annab tagasi pointeri tähele a sõnas "lause"..
- **char strstr(cs1, cs2)** Otsib stringist cs1 alamstringi cs2
  - Eelmised 4 annavad ebaedu korral kõik tagasi NULL
- **size_t strspn(cs1, cs2)** Tagastab cs1 algusest pikkuse, mis koosneb ainult cs2-s olevatest tähtedest.
  - nt:. strspn("Siil udus", “ilu S”) == 6
- **size_t strcspn(cs1, cs2)** Tagastab cs1 algusest pikkuse, milles ei esine cs2 tähti
  - nt strcspn("Siil udus", "abs") == 8

string.h failist leiate neid veel!

strpbrk()

- char *strpbrk(const char *s, const char *accept);
  - pbrk – pointer to break

```
char string[100] = "The 3 men and 2 boys ate 5 pigs\n";
 char *result;
 printf( "1: %s\n", string );
 result = strpbrk( string, "0123456789" );
 printf( "2: %s\n", result++ );
 result = strpbrk( result, "0123456789" );
 printf( "3: %s\n", result++ );
 result = strpbrk( result, "0123456789" );
 printf( "4: %s\n", result );
```

strspn()

- size_t strspn(const char *s, const char *accept);
  - ütleb mitu tähte stringist koosneb täielikult *accept*-i tähtedest

```
char string[] = "cabbage";
int  result;
result = strspn( string, "abc" );
printf( "The portion of '%s' containing only "
 "a, b, or c is %d bytes long\n", string, result );
```

strstr()

- char *strstr(const char *haystack,
  const char *needle);

```
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
char *pdest;    int  result;

printf( "String to be searched:\n\t%s\n", string );
printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
pdest = strstr( string, str );
result = pdest - string + 1;
if( pdest != NULL )
   printf( "%s found at position %d\n\n", str, result );
else
   printf( "%s not found\n", str );
}
```

typedef tüübid

- C-s on võimalus luua uusi andmetüübinimesid

```
typedef int Length;     /* Defineerimine */
Length len, arr[Size];  /* Kasutus */

typedef char* String; /* String on string */
```

typedef (2)

- typedef ei ole makro

```
const int BUF_SIZE = 8;

typedef char Buf[BUF_SIZE]; /* viga */
```

vs

```
#define BUF_SIZE 8;
const int SIZE=100;
typedef char Buf[BUF_SIZE];
Buf buffer, buff_arr[SIZE];
     /*NB buff_arr on kahemõõtmeline */
```

Milleks typedef

- Lihtne andmetüüpe vahetada
  - Näiteks kasutame tarkvaras *inti* pikkuseid lippe (*flag*). Programmi arendamise käigus läheb programm keerukamaks ja vajame pikemat muutujat: tahame, et tüüp oleks *long*. Kui oleksime deklareerinud tüübi Flag, oleks tarvis muuta ainult typedef rida.
- Loetavus ja dokumentatsioon
  - Andmetüüpidele tähenduslikud nimed
  - Eelmises näites oleks Flag var; reast kohe aru saada, et var on mitte mingi suvaline arv, vaid et seal sees on meie lipud hoiul.

Skoopimise reeglid

- Muutujanime saame kasutada vaid mingist programmiosast. Selle osa määrab ära muutuja **skoop**.
- Bloki skoop

```
1: {
       int k;
       ...
2:     {
                ....
       }
}
```

  - Muutujat k saame kasutada nii blokis 1, kui blokis 2. neist väljaspool seda teha ei saa. Samuti pole blokis 1 näha ühtki väljaspool deklareeritud k-d.

Skoopimise reeglid (2)

- Faili skoop

```
int k;
1: {
   ...
   2: {
      ...
   }
}
3: {
   ...
}
```

Kõik nimed, mis on deklareeritud väljaspool blokke on faili skoobis: seega on nad kättesaadavad kogu faili ulatuses.
Arvesta ka sellega, et #include käskudega sisse toodud muutujad käituvad nagu nad oleksid samas failis

Skoopimise reeglid (3)

- Prototüübi skoop:

```
   void func1(int count);
```

  See skoop kehtib ainult funktsiooniprototüübis kirjutatud funktsiooniargumentidele. Sealsed argumendinimed ei loe. Nad ei pea kattuma funktsiooni definitsioonis toodud nimedega (aga võiksid samas).

- Funktsiooni skoop:

```
void func1(int counter) {
   ...
}
```

  See skoop kehtib ainult funktsiooniblokis sees olles. ANSI C-s on funktsiooniskoop sisuliselt blokiskoop.

Näide "heast" programmeerimisest

```
{
  int a = 1, b = 2, c = 3;
  printf("%d %d %d\n", a, b, c);
  {
      int b = 4;
      float c = 5.0;
      a = b;
      {
            int c;
            c = b;
            printf("%d %d %d\n", a, b, c);
      }
      printf("%d %d %g\n", a, b, c);
  }
  printf("%d %d %d\n", a, b, c);
}
```

Terminitest

- Sisemine muutuja (*Internal variable)*: deklareeritud funktsiooni või bloki sees
  - Bloki või funktsiooni **sisesed**: neid pole näha väljaspool
  - Nad on reeglina ka **automaatsed**: tekkides blokki või funktsiooni sisenedes ja hävides sealt lahkudes
  - **Staatilised** muutujad luuakse ja initsialiseeritakse kompileerimise ajal. Nende väärtus säilib ka blokist lahkumisel, kuigi neid pole väljaspool näha.

```
static int my_count = 0;
my_count++;
```

Näide funktsioonist, mis muuhulgas mäletab mitu korda ta käivitati.

Terminitest (2)

- **Välised** või **Globaalsed** muutujad: muutujad, mis on deklareeritud väljaspool funktsioone ja mida saab kõigist funktsioonidest kasutada ja muuta.
  - säilitavad väärtuse programmi lõpuni
  - võimaldavad andmeid vahetada
  - funktsioon saab muutujat kasutada kui
    - see on failis varem defineeritud
    - on failis **`extern`** tähisega defineeritud
    - on funktsioonis **`extern`** tähisega tähistatud
    - NB! extern ütleb, et muutuja on "kuskil mujal" deklareeritud: mälu sellele ei reserveerita.

extern

```
/* file1.c */
int my_count = 0;
char *func1();
...
func1();
```

```
/* file2.c */
extern int my_count;
extern char *func1();
...
my_count++;
```

Deklaratsioon vs. Definitsioon

- Deklaratsioon on midagi, mis ütleb kompilaatorile lisainfot:
  - extern int x;
  - float square(float);
  - typedef int * Link;
- Kui deklaratsioon põhjustab ka mälu eraldamist, on ta definitsioon:
  - int x;
  - Link y;
  - Muutujat võib korra defineerida ja mitu korda deklareerida

Listinäide (list.h)

```
#ifndef _LIST_H
#define _LIST_H

struct listitem {
    int data;
    struct listitem *next;
};

typedef struct listitem Listitem;

struct list {
   Listitem *head;
};

typedef struct list List;

void initlist (List *);       /* initialize an empty list */
void insertfront(List * , int val); /* insert val at front */
void insertback(List *, int val);     /* insert val at back */
int length(List);                     /* returns list length*/
void destroy(List *);                 /* deletes list */
void setitem(List *, int n, int val); /* modifies item at n to
                                         val*/
int getitem(List, int n);             /* returns value at n*/
#endif /* _LIST_H */
```

Listinäide (list.c)

```
#include "list.h“

void initlist(List *ilist) {
  ilist->head = NULL;
}

void insertfront(List *ilist, int val) {
   Listitem *newitem;
   newitem = (Listitem
     *)malloc(sizeof(Listitem));
   newitem->next = ilist->head;
   newitem->data = val;
   ilist->head = newitem;
}
```

Listinäide (list.c) (2)

```
void insertback(List *ilist, int val) {
   Listitem *ptr;
   Listitem *newitem;
   newitem = (Listitem
     *)malloc(sizeof(Listitem));
   newitem->data = val;
   newitem->next = NULL;
   if (!ilist->head) {
      ilist->head = newitem;
      return;
   }
   ptr = ilist->head;
   while (ptr->next){
      ptr = ptr->next;
   }
   ptr->next = newitem;
}
```

Listinäide (list.c) (3)

```
int length(List ilist){ /* returns list length */
   Listitem *ptr;
   int count = 1;
   if (!ilist.head) return 0;
   ptr = ilist.head;
   while (ptr->next) {
      ptr = ptr->next;
      count++;
   }
   return count;
}
```

Listinäide (list.c) (4)

```
void destroy(List *ilist) { /* deletes list */
   Listitem *ptr1,*ptr2;
   if (!ilist->head) return;
                     /* nothing to destroy */
   ptr1 = ilist->head; /* destroy one by one */
   while (ptr1) {
      ptr2 = ptr1;
      ptr1 = ptr1->next;
      free(ptr2);
   }
   ilist->head = 0;
}
```

Listinäide (list.c) (5)

```
void setitem(List *ilist, int n, int val){
 /* modifies a value*/
 /* assume length is at least n long */

   Listitem *ptr;
   int count = 0;
   if (!ilist->head) return;
   ptr = ilist->head;
   for (count = 0;count < n;count ++) {
      if (ptr) ptr = ptr->next;
      else return;
   }
   if (ptr)
     ptr->data = val;
}
```

Listinäide (list.c) (6)

```
int getitem(List ilist, int n) {
   /* returns a list value,
    * assume length is at least n long */
   Listitem *ptr;
   int count = 0;
   if (!ilist.head) return 0;
   ptr = ilist.head;
   if (n==0) return ptr->data;
   while (ptr->next) {
      ptr = ptr->next;
      count++;
      if (n == count)
        return (ptr->data);
   }
   return 0;
}
```

Listinäide (main.c)

```
#include "list.h"
main ()
{
   List mylist;
   initlist(&mylist);
   printf("L=%d\n",length(mylist));
   insertback(&mylist,1);
   printf("L=%d\n",length(mylist));
   insertback(&mylist,2);
   printf("L=%d\n",length(mylist));
   insertback(&mylist,3);
   printf("L=%d\n",length(mylist));
   insertback(&mylist,4);
   printf("L=%d\n",length(mylist));
   printf("item 0 = %d\n",getitem(mylist,0));
   printf("item 1 = %d\n",getitem(mylist,1));
   printf("item 2 = %d\n",getitem(mylist,2));
   printf("item 3 = %d\n",getitem(mylist,3));
   destroy(&mylist);
}
```

Listinäide (väljund)

```
L=0
L=1
L=2
L=3
L=4
item 0 = 1
item 1 = 2
item 2 = 3
item 3 = 4
```

Union

- union on andmetüüp, mis võib (eri aegadel) hoida endas erinevaid andmeid.
- Andmeid hoitakse samal aadressil
- Deklareerimime nagu struct

```
union speed {
   int landspeed;
   long flightspeed;
   struct bizarre warpspeed;
} S;
```

- Kasutame samuti:
  - union.liige
  - unionipointer->liige

Union (2)

- Unioni võib ette kujutada structina, mille kõigi liikmete suhteline aadress on 0
- Parajasti hoitavate andmete tüübi meelespidamine on kasutaja probleem
- Initsialiseerimine on võimalik ainult esimese liikme tüübi korral
- Sizeof annab suurima hoitava tüübi numbri
- Võivad sisaldada struct-e või unione või neis ise sisalduda

Alignment

- Mõnd tüüpi saab protsessor ainult mingil kindlal aadressil hoida: juhuslikku kohta neid kirjutada pole mõtet
  - char 0, 1, 2, 3 jne
  - int 0, 4, 8 jne
  - double 0, 8, 16 jne
- Mälu hõivamisel ja pointeraritmeetika puhul tuleb arvesse võtta
- Structi puhul annab suuruse sizeof:

sizeof(a) >> 8

```
struct a_tag {
   char c;
/* 3 lisabaiti siin */
   int i;
} a;
```

Alignment (2)

- Malloc annab alati mälu, mis sobib kõigi andmetüüpide jaoks. St kui kõige suurem primitiivtüüp on 8 baiti, antakse mälu 8-ga jaguvalt aadressilt.

Bittoperaatorid

- Opereerivad numbritel bitikaupa, vaadeldes neid kahendkoodidena (mis õnneks on ka viis, kuidas neid arvutis hoitakse)
- 10ndsüsteemis ei ole nad eriti huvitavad, samas 2nd-, 8nd- ja 16ndsüsteemis on nad veidi sisukamad
- 8ndsüsteemi konstandi tegemiseks kirjutatakse konstandi ette lisanull ja kuueteistkümnendsüsteemi konstandi tegemiseks kirjutatakse ette 0x või 0X
- Nt 077 või 0xff

& = AND

- Üksik ampersand & teeb kahe arvu vahel bitthaaval AND tehet. Iga bitt on 1 siis, kui mõlemas arvus on vastava järgu bitid 1.
- 0x56 & 0x32 = 0x12

```
   01010110
 & 00110010
 ----------
   00010010
```

| = OR

- Püstkriips | teeb kahe arvu vahel bitthaaval OR tehet. Tulemuse vastava järgu bitt on 1 siis, kui ühes või teises arvus on vastavas järgus 1.
- 0x56 | 0x32 = 0x76

```
   01010110
 | 00110010
 ----------
   01110110
```

^ = XOR

- ^ operaator annab tulemuseks kahe arvu omavaheliste bittide XOR tehte. Bitt on 1 siis, kui mõlema arvu vastava järgu bitid on erinevad (1 juhul, kui ainult üks operaatoritest on 1)
- 0x56 ^ 0x32 = 0x64

```
   01010110
 ^ 00110010
 ----------
   01100100
```

~ = täiend

- Tilde ~ operaator annab arvu täiendi (complement). Tehe tehakse ainult ühe arvuga (On unaarne nagu !, & ("address of" tähenduses) ja *). Täiendiks on siis arv, kus on 1 ja 0 vahetatud.
- 16-bitise täisarvu puhul ~0x56 = 0xffa9

```
 ~0000000001010110
 ------------------
  1111111110101001
```

<< nihe vasakule

- << operaator nihutab esimese argumendi bitte teise argumendi jagu vasakule, täites uued kohad 0-dega.
- Välja nihkunud bitid heidetakse kõrvale
- 0x56<<2 = 0x58

```
000001010110 << 2
------------
000101011000
```

>> nihe paremale

- >> operaator nihutab esimest argumenti teise jagu paremale, täites vasakult poolt 0-dega.
- Juhul, kui tegemist on märgiga (*signed*) arvuga ja kõige vasakpoolsem bitt on 1, võib ta sisse nihutada ka 1. (Siit ka moraal: bitinihete puhul on mõistlik deklareerida muutujad märgita arvudena)
- 0x56 >> 1 = 0x2b

```
01010110 >> 1
--------
00101011
```

Nihete aritmeetiline tähendus

- Nihe vasakule << on sama, mis korrutamine 2-ga
- Analoogselt on 10ndsüsteemis nihe vasakule korrutamine 10-ga
- Nihe paremale >> on jagamine 2-ga, samas võib juhtuda, et säilitatakse kõige vasakpoolsem bitt (st nihutatakse sisse eelmise vasakpoolseimaga identne bitt).
- Nihe on loomulikult kiirem kui korrutamine või jagamine: Enamasti teeb asenduse kompilaator, seega pole tarvis keerukamat koodi kirjutada.

Ärgake

(Loeng sai läbi)